HIGH PERFORMANCE HARDWARE FOR MODULAR DIVISION/INVERSE

BY

MOHAMED ABUOBAIDA MOHAMED

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

25/5/2015

HIGH PERFORMANCE HARDWARE FOR MODULAR DIVISION/INVERSE

by

MOHAMED ABUOBAIDA MOHAMED

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

In Partial Fulfillment of the Requirements
for the degree

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

KING FAHD UNIVERSITY
OF PETROLEUM & MINERALS

Dhahran, Saudi Arabia

25/5/2015

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **MOHAMED ABUOBAIDA MOHAMED** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING**.

Thesis Committee

Dr. Alaaeldin Amin (Adviser)

(Co-adviser)

Dr. Sultan Almuhammadi (Member)

Dr. Ahmad Khayyat (Member)

(Member)

Dr. Ahmad Al-Mulhem
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

1/6/15
Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

To My Mother and Father

ACKNOWLEDGMENTS

First of all, I am thankful to Allah for giving me the ability to achieve this work. I am also grateful for my supervisor, Dr. Amin for all the guidance and support he gave me during my studies. In addition I thank the committee members, Dr. Almuhammadi and Dr. Khayyat for their assistance in making this possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME: Mohamed Abuobaida Mohamed

TITLE OF STUDY: High Performance Hardware for Modular Division/Inverse

MAJOR FIELD: Computer Engineering

DATE OF DEGREE: 25/5/2015

Computing the modular division or inverse is one of the most time consuming operations in several security applications. Performing the operation in hardware significantly improves the performance of such applications. Two algorithms are proposed in this work. The first performs modular division in GF(p). It is based on the Extended Euclidean and binary GCD algorithms. The second computes the Montgomery modular inverse based on a modified Kaliski modular inverse algorithm. The modification removes the final subtraction step. Parameterized hardware models for both algorithm were developed using VHDL. The models achieve constant cycle time independent of the operands sizes. The implementation of the division algorithm occupies nearly half the area and about 30% less critical path delay compared to previous work. The number of iterations of the inverse algorithm is 80% less than other constant cycle time implementations.

ملخص الرسالة

الاسم الكامل: محمد أبو عبيدة محمد

عنوان الرسالة: عتاد عالي الأداء لحساب القسمة النمطية والنظير الضربي النمطي

التخصص: هندسة حاسب آلي

تاريخ الدرجة العلمية: ٢٠١٥/٥/٢٥

حساب القسمة النمطية (Modular Division) أو النظير الضربي النمطي (Modular Inverse) يعد من أطول العمليات في العديد من التطبيقات الأمنية. تنفيذ هذه العمليات في العتاد يحسن أداء هذه التطبيقات بشكل كبير. هذا العمل يقدم خوارزميتين في هذا المجال. الأولى تقوم بحساب القسمة النمطية في حقول جالوا التي يكون مميزها عدد أولي، وهي مبنية على الخوارزمية الإقليدية الموسعة وخوارزمية القاسم المشترك الأكبر الثنائية. تقوم الخوارزمية المقدمة الثانية بحساب النظير الضربي النمطي في مجال مونتغومري. وهي نسخة معدلة من خوارزمية كاليسكي لحساب النظير الضربي النمطي في مجال مونتغومري. التعديل يزيل خطوة الطرح الأخيرة في خوارزمية كاليسكي. وقد تم تطوير نماذج لكلا الخوارزميتين باستخدام VHDL. يحقق النموذجين زمن دورة ثابت لا يعتمد على أحجام المعاملات المدخلة. نموذج القسمة يستخدم ما يقرب من نصف المساحة وحوالي ٣٠٪ أقل في زمن الدورة مقارنة بالأعمال السابقة. أما نموذج النظير الضربي فينهي العملية في عدد دورات أقل بـ ٨٠٪ من النماذج الأخرى التي تقدم زمن دورة ثابت.

CHAPTER 1

INTRODUCTION

Secure communication has become a major concern in recent years. Online trade, for example, requires the exchange of sensitive information like bank accounts and credit card numbers. Public key cryptosystems [1] were proposed to eliminate the need for a shared key between communicating parties. In these cryptosystems, the sender encrypts the message using a public key that is publicly announced. The message can be decrypted only knowing a private key which is mathematically related to the public key. Security of public key cryptosystems relies on the infeasibility of deducing the private key given the public key which is directly related to key lengths in bits. Longer keys provide higher security with current typical key lengths running in the thousands of bits. Examples of public key cryptosystems include: the Diffie-Hellman key exchange [1], ElGamal [2], RSA [3] and the Elliptic-Curve Cryptosystems (ECC) [4]. Most encryption and decryption operations in these cryptosystems involve modular arithmetic.

Another application of modular arithmetic is secret sharing and secret

reconstruction[5]. If a secret is to be stored, m different shares are generated out of this secret. Each of these shares does not have any significance by itself. The secret, however, may be reconstructed using any n ($n \leq m$) of these shares. The parameters m and n are set according to the required level of security.

Modular division and/or modular inverse is one of the most time consuming modular operations. It is an essential operation in many cryptosystems particularly in ECC and in secret sharing and reconstruction. In ECC, we can only use the simpler affine coordinates if the cost of implementing Inversion is reduced to a point where there will be no benefit at all using other coordinate system designed to avoid the inversion operation (e.g. projective or Jacobian) [6].

Therefore, building specialized hardware to compute the modular division will significantly enhance the performance of such systems. This work aims at producing an application specific hardware, which can efficiently perform modular division and inverse. The hardware targets improved speed or smaller area or both compared to other reported implementations.

This thesis is organized as follows: Chapter 2 reviews previous work. Chapter 3 introduces our proposed algorithm to compute modular division and its corresponding hardware. Our proposed implementation of the Montgomery modular inverse is described in Chapter 4. Conclusion and future work are given in Chapter 5.

CHAPTER 2

PREVIOUS WORK

There are two major methods for computing the result of modular division. The first is through performing the modular division directly and the second is through finding the modular inverse of the divisor first and then multiplying the result by the dividend. This chapter reviews the previous work in both approaches.

2.1 Review of Modular Division Algorithms

Work in the first method is typically used in prime fields $GF(p)$ and binary extension fields $GF(2^n)$. Elements in $GF(p)$ are the integers in the range 0 to $(p-1)$. Arithmetic operations in this field are performed modulo p. Elements of $GF(2^n)$ are polynomials the maximum degree of which is $n-1$ with coefficients of either 0 or 1. Addition and subtraction are equivalent in $GF(2^n)$, both are performed by XORing the corresponding coefficients of the operand polynomials [7]. When the degree of a resulting polynomial is greater than or equal to n, it is reduced modulo some selected irreducible polynomial.

Modular division algorithms are mostly based on the Extended Euclidean Algorithm (EAA) and the binary Greatest Common Divisor (GCD) algorithm and its extensions, e.g. the plus minus (PM) algorithm [8]. The Binary GCD algorithm makes use of the following relations:

- $GCD(A, B) = GCD(A - B, B)$
- If A is even and B is odd then $GCD(A, B) = GCD(A/2, B)$

The PM algorithm applies when both A and B are odd where 4 divides either $(A + B)$ or $(A - B)$, i.e, $4|(A + B)$ or $4|(A - B)$ and $GCD(A, B) = GCD((A \pm B)/4, B)$.The EEA and binary GCD algorithms are combined in the Extended Binary GCD Algorithm (algorithm 1)[8].

Input: X, Y, M; Where
$GCD(M, 2) = 1, 0 \leq X < M, 0 < Y < M, GCD(M, Y) = 1$
Output: $Z = X/Y \ mod \ M$

```
1  A = Y; B = M; U = X; V = 0;
2  while A > 0 do
3      if A mod 2 = 0 then
4          A = A/2; U = U/2 mod M;
5      else
6          if A ≥ B then
7              A = A − B; U = U − V;
8          else
9              swap(A, B); swap(U, V);
10         end if
11     end if
12 end while
13 Z = V;
14 return Z;
```

Algorithm 1: Extended Binary GCD Algorithm

Tawalbeh et.al. [9] presented a dual field modular division algorithm (algorithm 2). The algorithm computes modular division in both $GF(p)$ and $GF(2^n)$.

Numbers are represented in carry-save representation, with constant addition / subtraction delay. The core of the datapath uses a 4-2 compressor and a 3-2 adder. These adders are used for operations on the main registers A, B, U and V. The 3-2 adder is active only when registers A and B are both odd. To reduce the number of iterations, the algorithm uses a set of multiplexers to perform swapping. The algorithm has a fairly simple control flow due to the relatively few conditional statements.

The algorithm, however, has some drawbacks. It keeps track of the difference between the magnitudes of registers A and B through the parameter δ, which is a signed two's-complement binary number. Operations on δ, subtraction and negation, are integer operations with carry-propagation delay. The algorithm, further, uses two clocks in the same design for latching the outputs of the adders. Finally, due to the adopted carry-save representation, the loop termination check for $A = 0$ is quite a costly operation. They suggested performing this check in multiple clock cycles since the output value does not change after the algorithm is complete.

Kihara and Takagi [10] proposed a different approach (algorithm 3). They eliminated all operations that require carry propagation. Those operations were replaced by shift and set/reset operations, significantly reducing the loop iteration delay. The algorithm reduces the number of iterations by dedicating a hardware module to divide by 4 if $4|A$ rather than having to perform two successive divisions by 2 in two loop iterations. Compared to the carry-save representation

Input: $X, Y, M, Field$; Where $0 \leq X < M, 0 < Y < M, 2^{n-1} < M < 2^n$
Output: $Z = X/Y \ mod \ M$ when $Field = GF(p)$
$Z(x) = X(x)/Y(x) \ mod \ M(x)$ when $Field = GF(2^n)$

```
1  A = Y; U = X; B = M; V = 0; δ = 0;
2  while A ≠ 0 do
3      if a_0 = 0 then
4          A = A/2;
5          δ = δ − 1;
6      else
7          if δ < 0 then
8              swap(A, B);
9              swap(U, V);
10             δ = −δ;
11         end if
12         q = 1;
13         if A + B mod 4 ≠ 0 AND Field = GF(p) then
14             q = −1;
15         else
16             δ = δ − 1;
17         end if
18         A = (A + qB)/2;
19         U = (U + qV);
20     end if
21     U = (U + u_0 × M)/2;
22 end while
23 if B = 1 then
24     Z = V;
25 else
26     Z = M − V;
27 end if
28 return Z;
```

Algorithm 2: Dual Field Modular Division Algorithm

used by Tawalbeh et.al. [9], Kihara and Takagi used Binary Signed Digit (BSD) representation. While BSD has a constant addition time and simpler check for zero, it typically requires more logic per bit and hence more area compared to the carry-save representation.

While enjoying a very high speed, the Kihara-Takagi algorithm has some disadvantages. It contains many conditional statements resulting in a long iteration delay. In addition, detecting divisibility by 4 and performing this division consume a large area, because of the use of a dedicated modular quartering unit ($MQRTR(W, M)$).

A third algorithm (algorithm 4) for modular division was proposed by Chen et.al. [11]. They suggest systolic arrays to reduce carry propagation as opposed to using a redundant representation. Two systolic array architectures are suggested: interleaved and non-interleaved. The interleaved architecture has a hardware operating efficiency of 0.5, i.e., the hardware is idle for 50% of the time. It can perform two independent operations simultaneously in a way to achieve full hardware usage. However, it uses double the number of pipeline registers and requires twice the number of iterations compared to the non-interleaved architecture. The ratio of the critical path delay of the interleaved to that of the non-interleaved is approximately 0.7. The algorithm introduces a method to control the growth of U by using a flip-flop (p). When a modular reduction step is performed (steps 5, 13 and 17) p is negated.

Input: X, Y, M; Where $2^{n-1} < M < 2^n, GCD(M, 2) = 1$ and M is prime
$-M \leq X, Y < M, Y \neq 0$
Output: $Z = X/Y \ mod \ M, (-M < Z < M)$

```
1  A = Y; B = M; U = X; V = 0; P = 2^{n+1}; D = 1; s = 1;
2  while p_0 ≠ 1 do
3      if a_1 a_0 = 0 then
4          A = A/4; U = MQRTR(U, M);
5          if s = 0 then
6              if d_2 = 1 then s = 1;
7              if d_1 = 0 then D = D >> 2;
8              else P = P >> 1; s = 1;
9          else
10             D = D << 2;
11             if p_1 = 0 then P >> 2 else P >> 1;
12         end if
13     else if a_0 = 0 then
14         A = A/2; U = MHLV(U, M);
15         if s = 0 then
16             if d_1 = 1 then s = 1;
17             D = D >> 1;
18         else
19             D = D << 1; P = P >> 1;
20         end if
21     else
22         if a_1 a_0 + b_1 b_0 mod4 = 0 then q = 1;
23         else q = −1;
24         if s = 0 or d_0 = 1 then
25             A = (A + qB)/4; U = MQRTR(U + qV, M);
26             if s = 1 then
27                 P = P >> 1; D = D << 1;
28             else
29                 if d_1 = 1 then s = 1;
30                 D = D >> 1;
31             end if
32         else
33             {A = (A + qB)/4, B = A};  /* performed simultaneously */
34             {U = MQRTR(U + qV, M), V = U};
35             if d_1 = 0 then s = 0;
36             D = D >> 1;
37         end if
38     end if
39 end while
40 if b_1 b_0 = 3 or b_1 b_0 = −1 then V = −V;
41 return Z = V;
```

Algorithm 3: Hardware Algorithm for Modular Division

Input: X, Y, M; Where $|X| < M, 0 < |Y| < M, M < 2^n, GCD(Y, M) = 1$ and M is odd
Output: $Z = X/Y \ mod \ M$ and $|Z| < M$

```
1  A = Y; B = M; U = X; V = 0; φ = n; δ = 0; p = ±1;
2  while φ > 0 do
3      if a_0 = 0 then
4          A = A/2;
5          U = (U + p u_0 M)/2 mod M; p = −p;
6          if δ ≤ 0 then φ = φ − 1;
7          δ = δ − 1;
8      else                                          /* A is odd */
9          if (A + B) mod 4 = 0 then q = 1 else q = −1;
10         if δ ≥ 0 then
11             A = (A + qB)/2;
12             U = U + qV;
13             U = (U + p u_0 M)/2 mod M; p = −p;
14         else
15             {A = (A + qB)/2, B = A};  /* performed simultaneously */
16             {U = (U + qV), V = U};
17             U = (U + p u_0 M)/2 mod M; p = −p;
18             δ = −δ;
19         end if
20     end if
21 end while
22 if B ≡ 3 mod 4 then V = −V;
23 Z = V;
24 return Z;
```

Algorithm 4: Modular Division Algorithm for Systolic Implementation

2.2 Review of Modular Inverse Algorithms

The second major method for performing modular division is a two-step approach. The first step computes the modular inverse of the divisor and the second step performs modular multiplication of the result by the dividend. Since computing the modular inverse takes significantly more time compared to modular multiplication, it will be the focus of this review.

Montgomery [12] introduced an efficient method for multiplication in which the input integer operands are mapped to another domain (will be referred to as the Montgomery Domain). As conversion to and from the Montgomery domain is costly, Montgomery-based operations are most efficient when many operations are performed in succession.

Montgomery domain is defined by selecting a modulus M and a reference number r which is relatively prime to M. Typically, r is chosen as a power of 2 value (2^n) where n is the number of bits in M so as to simplify modular reduction. The image of a given integer number a in the Montgomery domain (denoted $\dot{a}$) is given by $\dot{a} = ar \ mod \ M$. The addition and subtraction operations are preserved in the Montgomery domain. Multiplication of two numbers $\dot{a}$, $\dot{b}$ in the Montgomery domain is defined as $\dot{c} = MM(\dot{a}, \dot{b}) = \dot{a}\dot{b}r^{-1} \ mod \ M$. The relation between the normal and the Montgomery domains is preserved with such definition of the Montgomery multiplication. If $c = ab$, $\dot{c} = \dot{a}\dot{b}r^{-1} = (ar)(br)r^{-1} = abr = cr \ mod \ M$. Montgomery multiplication is also used to convert numbers between the two domains. For example, $\dot{a} = MM(a, r^2) = ar^2r^{-1} = ar$ and

$a = MM(\dot{a}, 1) = arr^{-1} = a$.

Kaliski [13], introduced an algorithm to compute the Montgomery modular inverse of an operand a, which equals $a^{-1}2^n \ mod \ M$. The algorithm comprises two phases. The first phase computes $GCD(a, M)$ and postpones halvings related to modular inverse. The output of the first phase is $a^{-1}2^k \ mod \ M$ where $k - n$ is the number of postponed halvings. The second phase completes the algorithm by performing $k - n$ halvings of the result of the first phase.

Input: A, M, n; Where M is odd and $n = \|M\|$
Output: "Not Relatively Prime" or $Z = A^{-1}2^n \ mod \ M$

Algorithm First Phase:
$U = M$; $V = A$; $R = 0$; $S = 1$; $K = 0$;
while $V > 0$ **do**
    if $u_0 = 0$ **then**
        $U = U/2$;
        $S = 2S$;
    else if $v_0 = 0$ **then**
        $V = V/2$;
        $R = 2R$;
    else if $U > V$ **then**
        $U = (U - V)/2$; $R = R + S$; $S = 2S$;
    else
        $V = (V - U)/2$; $S = R + S$; $R = 2R$;
    end if
    $K = K + 1$;
end while
if $U \neq 1$ **then return** "Not Relatively Prime";
if $R > M$ **then** $R = R - M$;
Algorithm Second Phase:
for $i = 0$ **to** $K - n$ **do**
    if $r_0 = 0$ **then** $R = R/2$;
    else $R = (R + M)/2$;
end for
$Z = M - R$;
return Z;

Algorithm 5: Kaliski's Algorithm For Montgomery Modular Inverse

The number of iterations of the first phase in Kaliski's algorithm is at least n and at most $2n$. All intermediate values are between 0 and $2M$. These well-defined properties made the algorithm the basis for almost all following attempts to calculate the Montgomery modular inverse.

Savas and Koç [14] based their work on Kaliski's algorithm. They proposed a multiprecision approach where multiples of the computer's word ($m = i * w$) is used instead of the arbitrary number of bits in M. They modified the second phase of the algorithm to match their definition. They used Montgomery multiplication as a replacement to the repetitive halvings yielding 14% speedup over the original algorithm. By propoer choise of input operands of Montgomery multiplication, they managed to obtain the classical modular inverse and the Montgomery inverse for numbers in the Montgomery domain.

De Domale et.al. [15] proposed an FPGA oriented algorithm based on Kaliski's. They allowed the U, V registers to have negative values and used two's-complement representation. They removed the magnitude comparison step ($U > V$, step 10) and used sign detection to determine the correct operation to be performed. The proposed datapath suggests the use of carry select adders when the carry chain exceeds the FPGA's column height.

Naseer and Savas [16] replaced the magnitude comparison with bitsize comparison. Because bitsize comparison is not exact, they introduced a correction step if the result goes negative in addition to a final correction step if the result of the first phase is negative. The used adders that support operations in $GF(p)$

and $GF(2^n)$. They reported 70% speedup over the original algorithm.

Lórenz and Hlaváč [17] proposed an algorithm to calculate the Montgomery inverse that does not use subtractions. They based their work on the Savas and Koç's algorithm [14]. The main difference is that the U register is initialized with $-M$ instead of M. They showed that the modification does not affect the result or the number of iterations.

Deng and Zhou [18] worked on improving the first phase of Kaliski's algorithm. They reduced the number of iterations by checking for the loop condition $U \neq 1$ and $V \neq 1$ instead of $V > 0$. They also introduced radix-4 and radix-8 versions of the algorithm. The high-radix algorithms have more conditional statements and complex hardware implementation. However, they reported an overall speed up of 11.3% for the radix-4 algorithm (1024 bits) and up to 17.5% for the radix-8 algorithm (2304 bits). They found that the advantage of using high radix algorithm is more pronounced with large size operands.

Savas [19] introduced a carry-free architecture to compute the Montgomery inverse. He used Binary Signed Digit as the number representation and provided logic design for adders similar to the carry-save design. He restricted a unique representation of 0 by not allowing the negative and positive bits to be simultaneously 1. He used mask registers to keep track of operand sizes and provided a method to reduce the size if numbers grow larger than a defined limit. He ran a statistical analysis of the algorithm and showed that the average number of clock cycles is $12.34n$ were n is the bit length of the modulus M. Savas's algorithm

is a direct implementation of Kaliski's algorithm using BSD implementation. It assumes the availability of operand sizes as input parameters.

CHAPTER 3

ENHANCED MODULAR DIVISION ALGORITHM

In this chapter, the proposed division algorithm is presented, the algorithm's proof of correctness is given, and a hardware model is proposed. Finally, results and a comparison are discussed.

3.1 The Algorithm

Similar to earlier reported algorithms (Algorithms 2 and 3), the proposed algorithm is based on the Extended Euclidean Algorithm. The algorithm is implemented using the carry-save format, so as to obtain a constant iteration delay which is independent of the size of the operands. In addition, the carry-save adder's area-per-bit requirement is less than that of the binary signed-digit format.

The algorithm uses four operands (A, B, U and V). In each iteration, based on the value of A and B, an operation is selected to be performed on A and B as well as on U and V. An upper limit of the magnitude of A is stored in a_size. Similarly, b_size stores an upper limit of the magnitude of B. These registers are represented by shift registers to minimize the iteration delay. The difference $a\_size - b\_size$ is captured in the δ parameter. The value of δ is represented by a shift register D and a sign flag $s \in \{-1, 1\}$ to reduce the cycle time ($\delta = s \times \log_2(D)$). The algorithm is shown in algorithm 6. Table 3.1 highlights differences between the proposed algorithm and previously reported ones.

3.2 Mathematical Analysis

3.2.1 Proof of Correctness

The algorithm maintains the following congruence relations:

$$A \times X \equiv U \times Y \ mod \ M \tag{3.1}$$

$$B \times X \equiv V \times Y \ mod \ M \tag{3.2}$$

$$GCD(A, B) = 1 \tag{3.3}$$

This can be verified by induction:

- At initialization, $A = Y$ and $U = X$. Therefore, equation 3.1 holds. $B = M$ and $V = 0$ therefore equation 3.2 holds. Both equations hold after

Input: X, Y, M; Where $0 \leq X < M, 0 < Y < M$, M is odd prime
Output: $Z = X/Y \ mod \ M$

```
1  A = Y; B = M; U = X; V = 0;
2  a_size = ||Y||; b_size = ||M||; D = 2^(||M||−||Y||); s = −1;
3  while a_size ≠ 0 do
4      if a_0 = 0 then
5          A = A/2;
6          a_size = a_size >> 1;
7          if s = 1 then                                   /* δ > 0 */
8              D = D >> 1;                                 /* δ = δ − 1 */
9              if d_0 = 1 then
10                 s = −1;
11             end if
12         else                                            /* δ ≤ 0 */
13             D = D << 1                                  /* δ = δ + 1 */
14         end if
15     else                                                /* A is odd */
16         if d_0 = 0 AND s = −1 then                      /* δ < 0 */
17             A ⇔ B; U ⇔ V; a_size ⇔ b_size; s = 1;
18         end if
19         q = 1;
20         if [a_1 a_0] + [b_1 b_0] ≠ 0 then               /* A + B mod 4 ≠ 0 */
21             q = −1;
22         end if
23         A = (A + qB)/2;
24         U = (U + qV);
25     end if
26     e = opp_sign(U);          /* e = 1 when estimated sign of U is
       negative, −1 otherwise */
27     U = (U + e × u_0 × M)/2;
28 end while
29 if B = 1 then
30     Z = V;
31 else
32     Z = −V;
33 end if
34 return Z;
```

Algorithm 6: Enhanced Modular Division Algorithm

Table 3.1: Comparing Algorithm 6 with Algorithms 2 and 3

Algorithm	Algorithm 6	Algorithm 2	Algorithm 3
Radix	2	2	4
Number Representation	Carry-Save	Carry-Save	Signed-Digit
δ implementation	Set/reset and shift	Integer Arithmetic	Set/reset and shift
Loop Ending Condition	Based on size	Comparison against 0	Based on size
Initial Size	Pre-computed	None	Assumes full size

initialization regardless of the values of X, Y and M.

- When A is even, the algorithm performs $A = A/2$ (step 5) and $U = U/2$ (step 27) dividing both sides of equation 3.1 by 2.

- When A is odd, the algorithm computes $A = (A \pm B)/2$ (step 23) and $U = (U \pm V)/2$ (steps 24 and 27) changing the left hand side of equation 3.1 to $(A \pm B)/2 \times X = (A \times X \pm B \times X)/2$. By using equations 3.1 and 3.2 as the induction hypothesis, $(A \times X \pm B \times X)/2 \equiv (U \times Y \pm V \times Y)/2 = (U \pm V)/2 \times Y \ mod \ M$. Substituting $U = (U \pm V)/2$ in the right hand side of equation 3.1 returns the same final result as the left hand side. Therefore equation 3.1 holds.

- The algorithm preforms a conditional swap operation (step 17). Due to the similarity of equations 3.1 and 3.2, the swap operation keeps both equations holding.

Equation 3.3 is maintained throughout the algorithm since all operations are defined by the extended binary GCD algorithm (Algorithm 1).

After every iteration, the algorithm reduces the magnitude of A until $A = 0$. By equation 3.3, when $A = 0$, $B = \pm 1$. Equation 3.2 becomes: $\pm X \equiv V \times$

$Y \ mod \ M$. This means that $\pm V \equiv X \times Y^{-1} \ mod \ M$.

3.2.2 Numerical Example

The example given in Table 3.2 shows the result of $140/44 \ mod \ 151$ in 8 bits. The final result is $Z = V = 113$ since $B = 1$. Note that if a swap is performed, the next operation, $(A \pm B)/2$, is done on swapped operands.

Table 3.2: Numerical Example of algorithm 6

Operation	A	B	U	V	δ	$\log_2(a\_size)$	$\log_2(b\_size)$
Initialization	44	151	140	0	-2	6	8
$A >> 1$	22	151	70	0	-3	5	8
$A >> 1$	11	151	35	0	-4	4	8
swap, $(A - B)/2$	70	11	58	35	4	8	4
$A >> 1$	35	11	29	35	3	7	4
$(A - B)/2$	12	11	-3	35	3	7	4
$A >> 1$	6	11	74	35	2	6	4
$A >> 1$	3	11	37	35	1	5	4
$(A - B)/2$	-4	11	1	35	1	5	4
$A >> 1$	-2	11	76	35	0	4	4
$A >> 1$	-1	11	38	35	-1	3	4
swap, $(A - B)/2$	6	-1	74	38	1	4	3
$A >> 1$	3	-1	37	38	0	3	3
$(A - B)/2$	2	-1	75	38	0	3	3
$A >> 1$	1	-1	113	38	-1	2	3
swap, $(A + B)/2$	0	1	151	113	1	3	2
$A >> 1$	0	1	151	113	0	2	2
$A >> 1$	0	1	151	113	-1	1	2
$A >> 1$	0	1	151	113	-2	0	2
$140/44 \equiv 113 \ mod \ 151$							

3.2.3 Complexity Analysis

The algorithm completes after a_size reaches 0. The analysis is based on the a_size and b_size values and assumes operations in n bits. Initially, a_size and

b_size receive the number of bits in Y and M, respectively. Since $2^{n-1} < M < 2^n$, b_size initially equals 2^n while the initial a_size value ranges from 1 if $Y = 1$ to 2^n if $\|Y\| = \|M\|$.

For worst case analysis, let $a\_size = b\_size = n$. The worst case occurs when the algorithm goes through a swap operation after every division by 2. This results in $4(n + 1)$ iterations. The first part of the path is shown in Table 3.3.

Table 3.3: Worst Case of algorithm 6

Operation	Step	$\log_2(a\_size)$	$\log_2(b\_size)$	δ
Initialization	2	n	n	0
$(A \pm B)/2$	23	n	n	0
$A >> 1$	5	$n-1$	n	-1
swap, $(A \pm B)/2$	17, 23	n	$n-1$	1
$A >> 1$	5	$n-1$	$n-1$	0

3.3 Hardware Description

Registers

For the algorithm main loop to have a constant delay independent of the size of input operands, the carry-save format is used for all add/subtract operations in the loop. This requires that each of the operands (A, B, U and V) be represented using two registers; a sum register and a carry register. The A and U registers are updated each iteration while B and V are updated only after a swap operation.

The operands A, B, U and V are signed and can hold a value up to M. Therefore, their minimum size is $n+1$, where the extra bit is for the sign. Although the values of the operands are less than M, the individual components (sum and

carry) can be greater than M. Therefore, the sizes are expanded to $n + 2$ to capture any extra bits.

Compressors

Arithmetic operations in the algorithm are addition, subtraction and negation. These are performed using two 4-2 compressors (Comp_A and Comp_U) and one 3-2 carry-save adder (CSA_U). The implementation of algorthim 6 takes one cycle per iteration. Comp_A computes the next value of A and the combination of Comp_U and CSA_U finds the next value of U. To reduce the multiplexing delay, generating the final result Z from register V is performed in a dedicated CSA named CSA_V.

Sign Estimation of U

In every algorithm iteration, the U sum and carry registers (U_s and U_c) are updated with the equivalent of $U/2 \; mod \; M$ (step 27). If U is even, that is a direct right shift of U. If U is odd, then $\pm M$ is added to make the result divisible by 2. The choice of either $+M$ or $-M$ does not affect the correctness of the algorithm. However, repeated addition or subtraction can lead to overflow of the U registers. To avoid the overflow, a reasonable constraint to enforce is:

$$-M < U < M \tag{3.4}$$

If A is even, $U = (U \pm M)/2$ (step 27) is performed only. The choice between

$+M$ and $-M$ has no effect since $-M < (U \pm M)/2 < M$. If A is odd, $U = U \pm V$ (step 24) is performed then the division by 2 (step 27) is calculated. This operation can violate constraint 3.4 making $-2M < U < 2M$.

The chosen approach to meet the constraint is by checking the sign of U. If the sign of U is positive, $-M$ is selected otherwise $+M$ is selected. However, exact sign detection is not an efficient operation in carry-save format because it requires full carry-propagate addition. Therefore, the sign is estimated by adding, with carry propagation, the most significant bits of U_s and U_c.

To determine the minimum number of bits to add, consider operations in n bits. If t bits were used to estimate the sign of U, the source of error (ε) is the discarded least significant $n - t$ bits from the sum and carry components. If all these bits are 0, $\varepsilon = 0$ and if all are 1, $\varepsilon = 2 \times (2^{n-t} - 1)$. Therefore the range of error is: $0 \leq \varepsilon \leq 2^{n-t+1} - 2$.

The size of U_s and U_c is $n+2$ bits as stated earlier. By enumerating all possible cases, the minimum number of bits that meet constraint 3.4 is 3. To simplify the analysis, let $K = 2^{n-1}$ and sum be the value of adding the 3 most significant bits of U_s and U_c. Since $2^{n-1} < M < 2^n$, $K < M < 2K$ and when $t = 3$, $\varepsilon_{max} = 2K - 2$. Constraint 3.4 becomes $-K < U < K$ in the case of M_{min} and $-2K < U < 2K$ in the case of M_{max}. If the initial value of U meets constraint 3.4, by induction the next value of U also meets the constraint. These cases are shown in Table 3.4. Cells marked with $(-)$ are not possible since they infer an initial U outside the range of M_{min}. For the same reason, overflow is not possible

when $sum = 3$ and the maximum value of U when $sum = 3$ is $4K - 1$.

Table 3.4: Number of Bits in Sign Estimation of U

sum	Range of U	Selected Operation	Result Range	
			M_{min}	M_{max}
-4	$[-4K, -2K - 2]$	$(U + M)/2$	–	$(-K, -1)$
-3	$[-3K, -K - 2]$	$(U + M)/2$	–	$(-K/2, K/2 - 1)$
-2	$[-2K, -2]$	$(U + M)/2$	$(-K/2, K/2 - 1)$	$(0, K - 1)$
-1	$[-K, K - 2]$	$(U + M)/2$	$(0, K - 1)$	$(K/2, 3K/2 - 1)$
0	$[0, 2K - 2]$	$(U - M)/2$	$(-K/2, K/2 - 1)$	$(-K, -1)$
1	$[K, 3K - 2]$	$(U - M)/2$	$(0, K - 1)$	$(-K/2, K/2 - 1)$
2	$[2K, 4K - 2]$	$(U - M)/2$	–	$(0, K - 1)$
3	$[3K, 4K - 1]$	$(U - M)/2$	–	$(K/2, K - 1/2)$

3.3.1 The Datapath

Direct Hardware Implementation

The datapath is divided into two separate paths: the A, B path and the U, V path. The control signals (q, a_odd and $swap$) are calculated from the A, B path and applied to both paths. However, computing these signals in the beginning of the cycle result in a critical path that includes these computations. Therefore, we precompute them in the previous cycle. This implementation is shown in Figure 3.1 and Figure 3.2.

The output of Comp_A is always even. This means the least significant bits (LSBs) of the output are either both 0 or both 1. In the former case, the result can be shifted right without an issue. In the latter case, the two LSBs are added to the result in the next cycle after division by 2. The same is true for CSA_U.

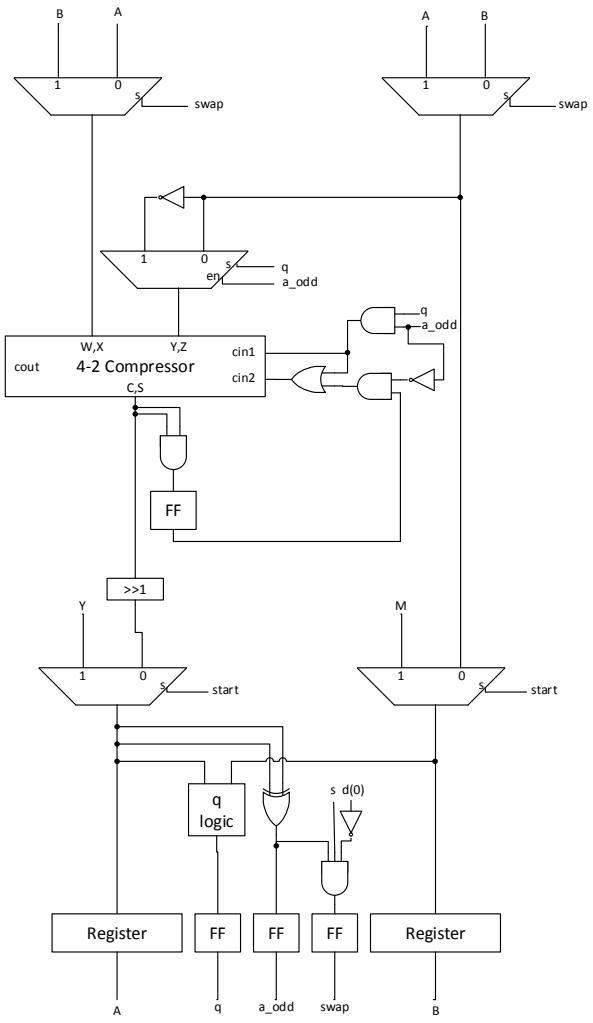

Figure 3.1: Direct A,B Path

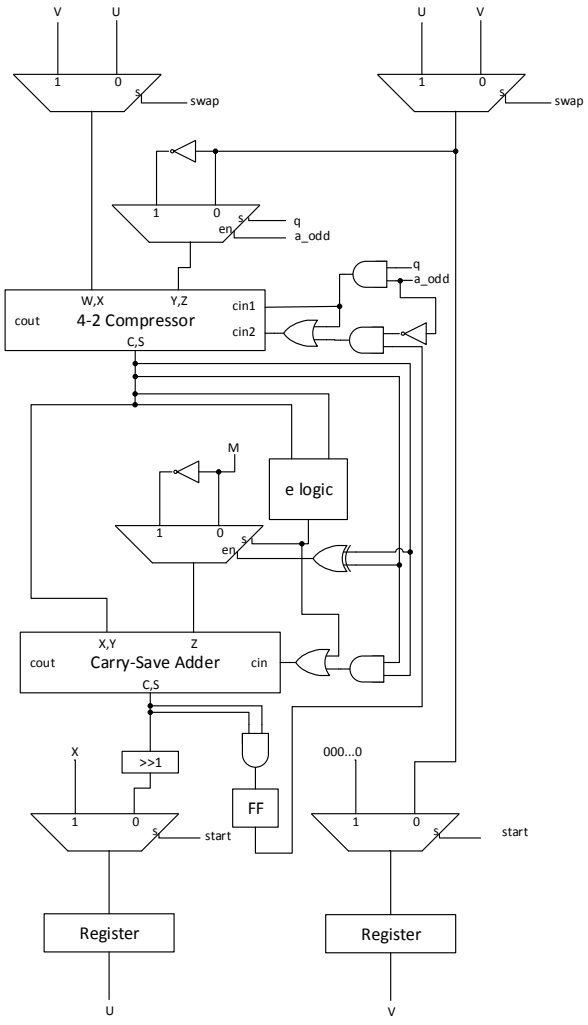

Figure 3.2: Direct U,V Path

Delayed Swapping

In the direct hardware implementation, swapping is performed before supplying the operands to the compressor using a multiplexer for every compressor input. This can be avoided by delaying the swap after the compressor. For A and U, swapping is done by inverting the output of Comp_A and Comp_U respectively. In the case of $A + B$, the operation is commutative, therefore there is no change required for swapping. For $A - B$, the result is negated by inverting all bits of the compressor outputs. The addition of LSBs for negating the outputs can be avoided by not adding the LSBs that are required for subtraction. The proof is shown next. It is based on the relation $-X = \overline{X} + 1 \Leftrightarrow \overline{X} = -(X + 1)$. The addition is performed in a 4-2 compressor, therefore the computation result has two components C and S. This implementation is shown in Figure 3.3 and Figure 3.4. This implementation removes one multiplexer from the critical path.

$$
\begin{aligned}
A_s + A_c + \overline{B_s} + \overline{B_c} &= A_s + A_c - B_s - 1 - B_c - 1 \\
&= A_s + A_c - B_s - B_c - 2 = C + S \\
\overline{C} + \overline{S} &= -(A_s + A_c - B_s - B_c - 2) - 1 - 1 \\
&= B_s + B_c - A_s - A_c + 2 - 2 \\
&= B - A
\end{aligned}
$$

Pre-Shifting

In order to remove the deferred addition of an LSB and to restrict the range of

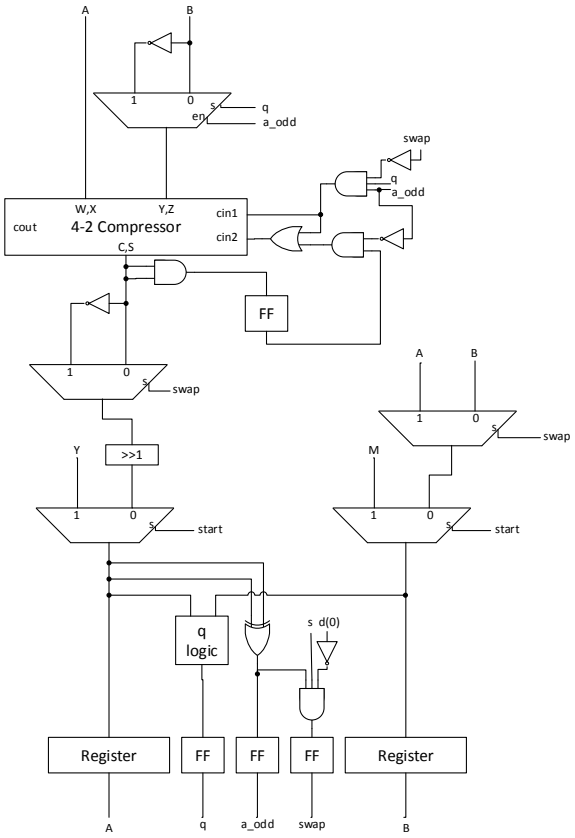

Figure 3.3: A,B Path with Delayed Swapping

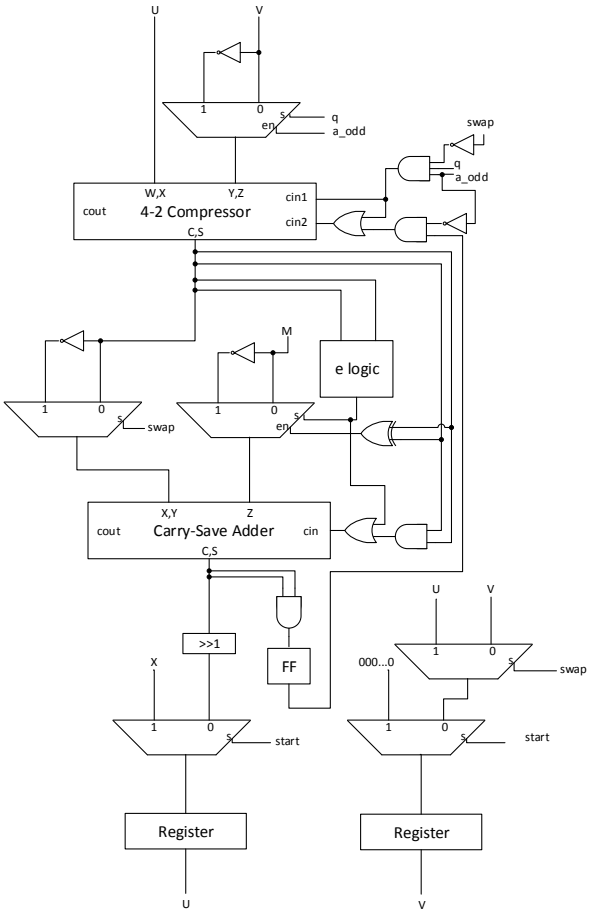

Figure 3.4: U,V Path with Delayed Swapping

the compressor output, division by 2 (steps 5, 23 and 27) can be achieved by right pre-shifting of the carry-save components A, B, U, V and M prior being applied at the inputs of the compressors. As this may result in the loss of the LSBs of these operands, this loss is compensated for by supplying proper carry inputs to the compressors. Furthermore, in case of subtraction, two LSBs should be added. The A,B path and U,V paths are discussed next and shown in Figure 3.5 and Figure 3.6.

The A,B path performs $A/2$ (step 5) when A is even or $((A \pm B)/2)$ (step 23), with a conditional swap operation (step 17), when A is odd. In the first case, the LSBs of A_s and A_c (a_s_0 and a_c_0) are either both 0 or both 1. When both are 0, Comp_A carry ins are both 0, otherwise Comp_A $cin2$ is set to 1. When A is odd and operation is addition, the sum $a\_s_0 + a\_c_0 + b\_s_0 + b\_c_0$ equals 2 because B is always odd. Therefore, after dividing the result by 2, a single 1 should be added and it selected to be in $cin2$. In the case of subtraction the sum $a\_s_0 + a\_c_0 - b\_s_0 - b\_c_0$ equals 0 indicating no loss of significant bits. However, two LSBs are added ($cin1 = cin2 = 1$) to achieve the negation of both components of B when swap is not performed. If there is a swap operation no LSBs are added as discussed in the delayed swapping implementation. These cases are summarized in Table 3.5.

Table 3.5: Comp_A Carry Inputs

Operation	$cin1$	$cin2$
$A/2$	0	a_s_0
$(A+B)/2$	0	1
$(A-B)/2$, no swap	1	1
$(A-B)/2$, swap	0	0

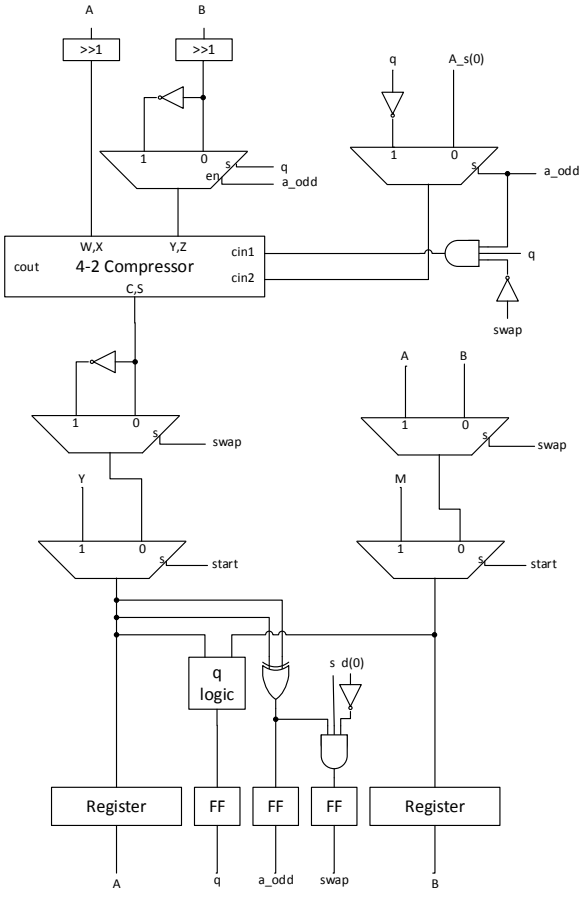

Figure 3.5: A,B Path with Pre-Shifting

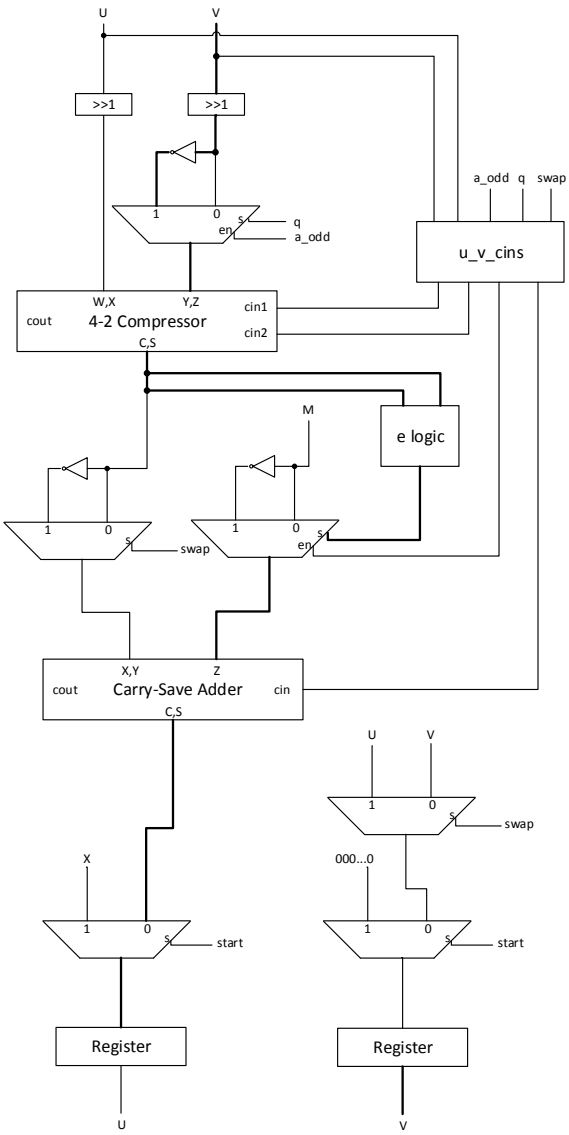

Figure 3.6: U,V Path with Pre-Shifting

The U,V path performs analogous operations to the A,B path namely $U/2$, $(U+V)/2$ and $(U-V)/2$. However the result might not be divisible by 2. In that case, $\pm M/2$ is added to the result to ensure the result is even. Combined with swapping, the path performs one of 12 operations every algorithm iteration. There are three carry inputs included in the the path, two in Comp_U and one in CSA_U. The value of U can be odd or even. When U is odd, $u\_s_0 \neq u\_c_0$ resulting in a unique case $u\_s_0 + u\_c_0 = 1$ (U_odd). When U is even it can be in one of two forms: $u\_s_0 = u\_c_0 = 0 \Rightarrow u\_s_0 + u\_c_0 = 0$ referred to as U_even_0 and $u\_s_0 = u\_c_0 = 1 \Rightarrow u\_s_0 + u\_c_0 = 2$ referred to as U_even_1. The same goes for V defining V_odd, V_even_0 and V_even_1. The operations of the U,V path are discussed next.

- $U/2$

  When doing this operation U is even and A is even. A single LSB is added when U_even_1 and none when U_even_0.

- $(U \pm M)/2$

  One of these operation occurs if U is odd and A is even. If $(U+M)/2$ is performed, the sum $u\_s_0 + u\_c_0 + m_0$ equals 2 indicating a single LSB to be added. If $(U-M)/2$ is performed, the sum $u\_s_0 + u\_c_0 - m_0$ equals 0 indicating no LSB to be added, however an LSB will be required for negation of M since M is represented in non-redundant form. This shows that both adding or subtracting M require adding a single LSB.

- $(U+V)/2$

  This operation indicates that the result of adding U to V is even. It can be inferred that U and V are either both odd or both even. The number of LSBs to add are summarized in Table 3.6. Entries marked with $-$ are not possible if this operation is performed.

Table 3.6: Number of Carry Inputs in U,V Path for Operation $(U+V)/2$

	V_odd	V_even_0	V_even_1
U_odd	1	–	–
U_even_0	–	0	1
U_even_1	–	1	2

- $(U+V\pm M)/2$

  Here $U+V$ is odd meaning one of U and V is even and the other is odd. Note that the addition or subtraction of M requires one LSB to be added. The results are in Table 3.7.

Table 3.7: Number of Carry Inputs in U,V Path for Operation $(U+V\pm M)/2$

	V_odd	V_even_0	V_even_1
U_odd	–	1	2
U_even_0	1	–	–
U_even_1	2	–	–

- $(U-V)/2$, no swap

  This operation is performed if the result of $U-V$ is even implying U and V are either both odd or both even. In addition, two LSBs are added to negate V. Because of negation, v_s_0 and v_c_0 carry negative weight. The results are shown in Table 3.8.

Table 3.8: Number of Carry Inputs in U,V Path for Operation $(U - V)/2$

	V_odd	V_even_0	V_even_1
U_odd	2	–	–
U_even_0	–	2	1
U_even_1	–	3	2

- $(U - V \pm M)/2$, no swap

  One of U and V is odd and one is even since $U - V$ is odd. The LSBs of V carry negative weight, two LSBs are added for negation and one for adding or subtracting M. The results are given in Table 3.9.

Table 3.9: Number of Carry Inputs in U,V Path for Operation $(U - V \pm M)/2$

	V_odd	V_even_0	V_even_1
U_odd	–	3	2
U_even_0	2	–	–
U_even_1	3	–	–

- $(U - V)/2$, swap

  The swap operation is performed by inverting the output of Comp_U. This makes the weight of its carry inputs negative. Also, the signs of U and V are reversed making u_s_0 and u_c_0 carry negative weight. No LSBs are added for negation as discussed in the delayed swapping implementation. The number of carry ins are shown in Table 3.10. The case U_even_1 and V_even_0 is marked with -1 signifying that this LSB has to be added in Comp_U not in CSA_U.

- $(U - V \pm M)/2$, swap

  Similar to the previous operation, LSBs of U have negative weights, no LSBs

Table 3.10: Number of Carry Inputs in U,V Path for Operation $(U - V)/2$, swap

	V_odd	V_even_0	V_even_1
U_odd	0	–	–
U_even_0	–	0	1
U_even_1	–	−1	0

are required for negation, one LSB is added for addition or subtraction of M. The summary is in Table 3.11.

Table 3.11: Number of Carry Inputs in U,V Path for Operation $(U - V \pm M)/2$, swap

	V_odd	V_even_0	V_even_1
U_odd	–	0	1
U_even_0	0	–	–
U_even_1	0	–	–

3.3.2 Algorithm Control

The algorithm control is a set of registers that defines the flow of the algorithm. The set includes D, s, a_size and b_size. D and s are used to represent $\delta = a\_size - b\_size$. They are related to δ by the relation: $\delta = s \times \log_2(D)$. D is modeled as a shift register with a single bit set to 1 and s is a single flip-flop. The value stored in D grows larger after repetitive execution of $A = A/2$ (step 5). Its largest value occurs when $A = 2^{n-1}$, where D will be shifted $n-1$ times (the high bit is shifted from bit position $n-1$ to 0). Therefore, the size of D is n.

The other two operands, a_size and b_size, represent the number of bits in A and B respectively. The $size$ is a register with a single bit set to 1 that denotes the location of the most significant bit of the operand it refers to. To reduce the area requirement, $size$ is modeled with two registers: $pattern$ (p-

bits) and *count* (*cs*-bits). These values, *size*, *pattern* and *count* are related by $\log_2(size) = \log_2(pattern) \times 2^{cs} + count$. Table 3.12 gives an example for this relation when $\|size\| = 8$, $p = 2$ and $cs = 2$. The initial value of *count* is found by encoding 2^{cs} bits corresponding to the location of the high bit in *pattern*.

The choice of p and cs can vary but it has to satisfy the relation $n = p \times 2^{cs}$. The choice affects the cycle time since *count* is a down counter with its delay dependent on its size. The optimal choice is when the delay of the counter is comparable to the delay of the main datapath.

Table 3.12: Relation Between *size*, *pattern* and *count*

$\log_2(size)$	$\log_2(pattern)$	*count*
0	0	0
1	0	1
2	0	2
3	0	3
4	1	0
5	1	1
6	1	2
7	1	3

The initial value of *pattern* is found by p or trees each taking n/p inputs from the input operand. If any tree produces a 1, outputs of next trees are disabled. The result is a single 1 in *pattern* corresponding to the n/p bits having the most significant bit of the operand. Figure 3.7 depicts this process.

To find initial *count*, the input operand is divided into p segments each of size n/p. Each segment is connected to a bus through a tri-state buffer. Each buffer is enabled be the corresponding *pattern* bit. The bus contains the segment containing the most significant bit of the operand. It is then supplied to a priority

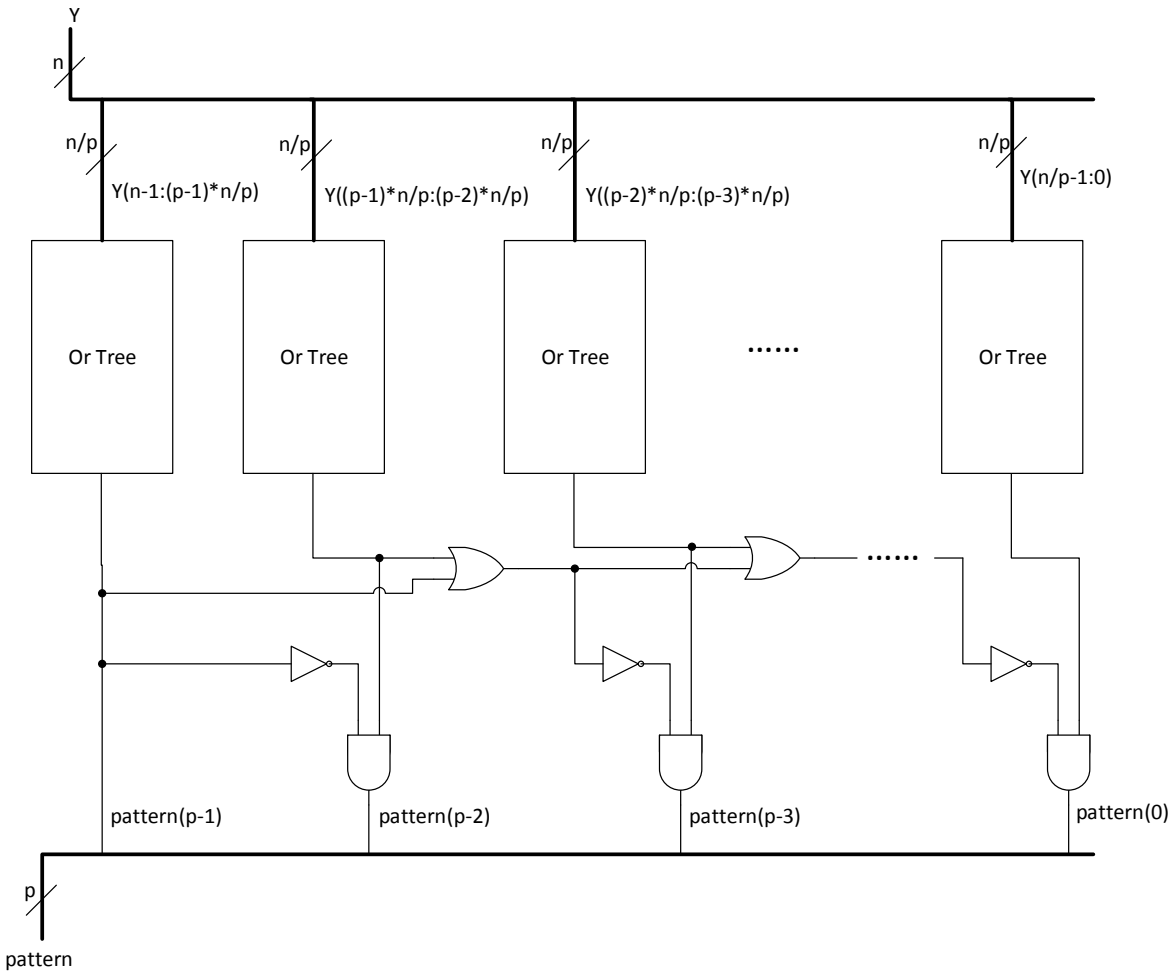

Figure 3.7: Pattern Finder

encoder to find the value of *count*. The diagram is shown in Figure 3.8.

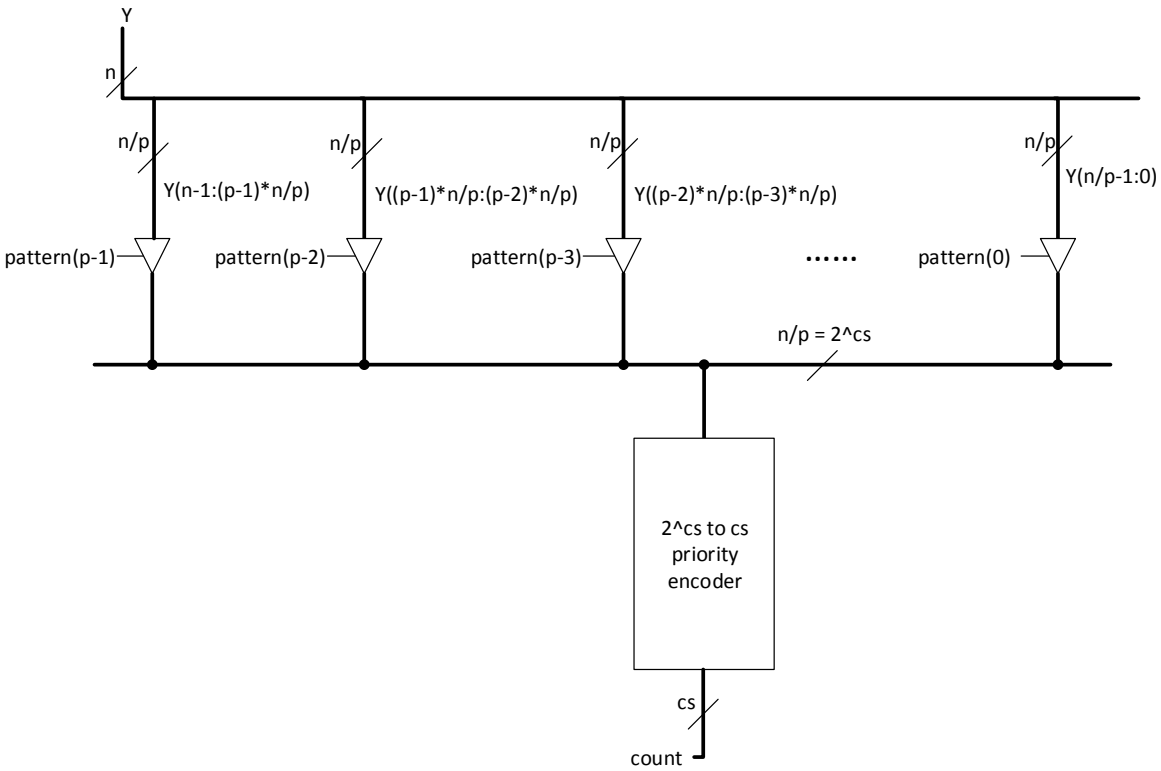

Figure 3.8: Count Finder

3.4 Implementation Results and Comparison

3.4.1 Number of Cycles

To get the average number of iterations, algorthim 6 was simulated in Java and compared to algorithm 3. The simulation was done for six moduli sizes (128, 192, 224, 256, 384, 512). For every modulus size, 15000 random divisions were performed. The same test cases were applied for both algorithms and the results are shown in Figure 3.9. Since both algorithms estimate the ending condition, both have an extra number of cycles. These are shown in Figure 3.10. The

ratios of the average number of cycles and extra cycles of algorthim 6 to those of algorithm 3 are presented in Figure 3.11. The average percentage of extra cycles in both algorithms is in Figure 3.12.

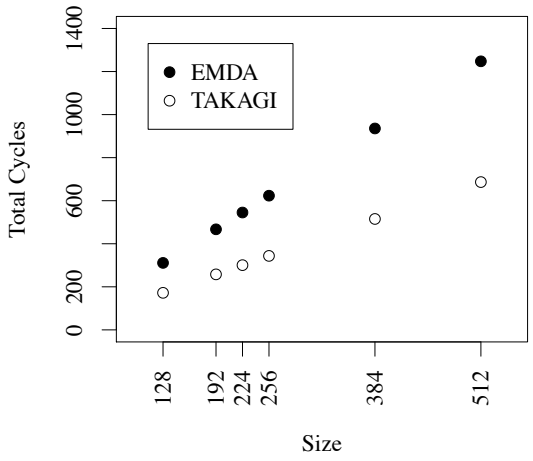

Figure 3.9: Average Number of Cycles

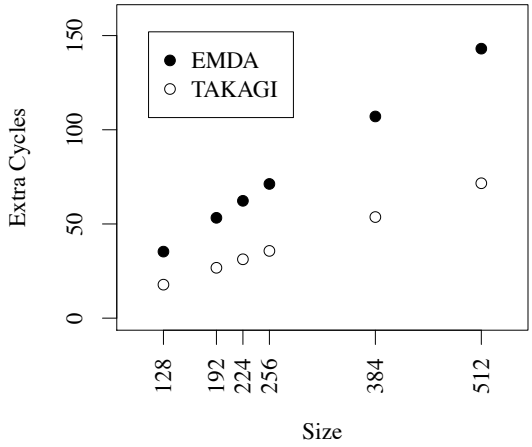

Figure 3.10: Average Number of Extra Cycles

The number of cycles in algorthim 6 is almost double the number of cycles in algorithm 3. This is expected since algorithm 3 checks and performs divisibility by 4 while algorthim 6 performs division by 2. The extra number of cycles are justified by the same reason. For both algorithms, extra cycles make slightly more than 10% of the total number of cycles. This similarity indicates that the two algorithms have almost equivalent ending criterion.

3.4.2 Synthesis Results

The algorithm was modeled in VHDL and simulated exhaustively for all 8-bit moduli to confirm correctness. The synthesis was performed with Synopsis Design

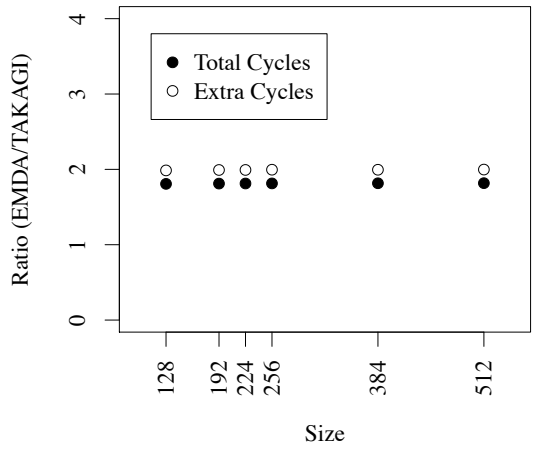

Figure 3.11: Comparing EMDA with Takagi

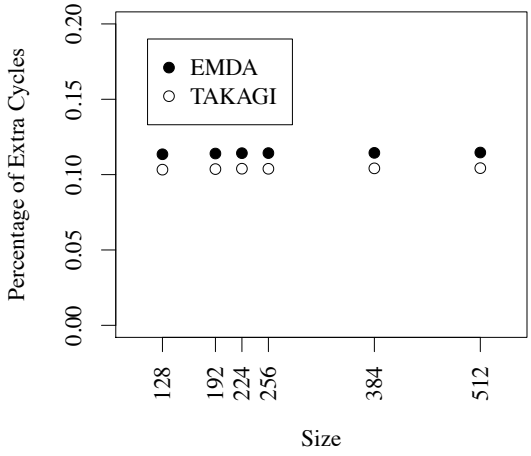

Figure 3.12: Average Percentage of Extra Cycles

Compiler with the ungroup option set. The target library was set to LFoundary 150 nm (PDK_LF150i_V2_0_0). The comparison is done against algorithm 3 which was modeled and synthesized using the same tools. The comparison was done for six moduli sizes (128, 192, 224, 256, 384, 512).

The critical path delay is shown in Figure 3.13 and the ratio in Figure 3.14. These figures show that algorthim 6 has smaller critical path. This confirms the simplicity of the control path in algorthim 6 compared to algorithm 3. Area comparison (Figure 3.15 and Figure 3.16) shows greater advantage for algorthim 6 over algorithm 3 with almost half the occupied area. This is due to the use of carry-save format which has smaller adder sizes. Figures 3.17, 3.18, 3.19 and 3.20 give the area * delay and area * delay^2 for both algorithms and the ratios between them.

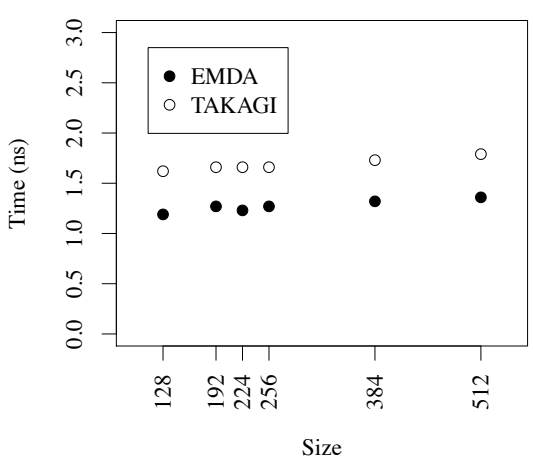

Figure 3.13: Critical Path Delay

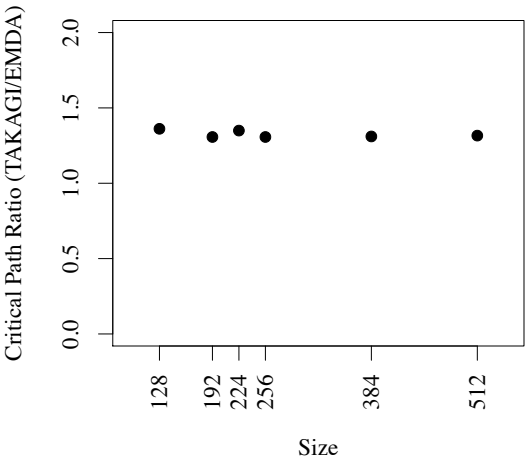

Figure 3.14: Critical Path Ratio

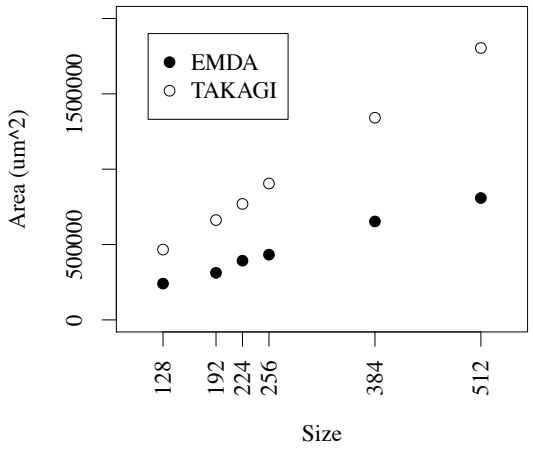

Figure 3.15: Area

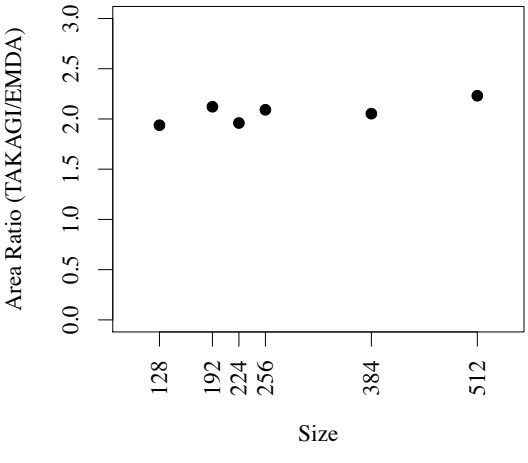

Figure 3.16: Area Ratio

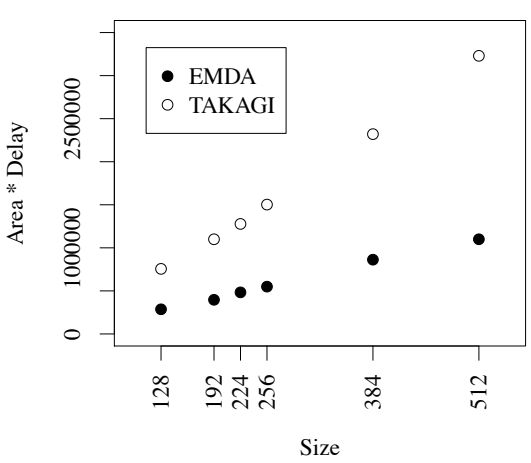

Figure 3.17: Area * Delay

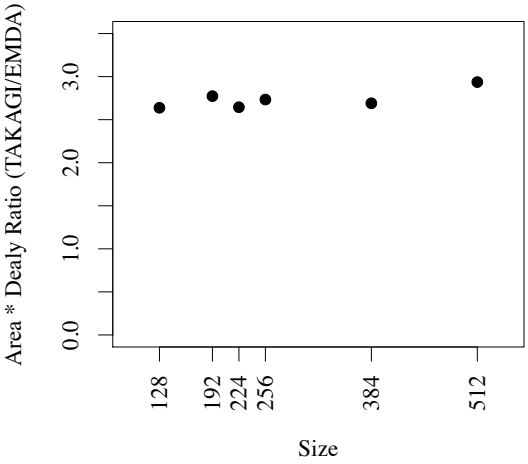

Figure 3.18: Area * Delay Ratio

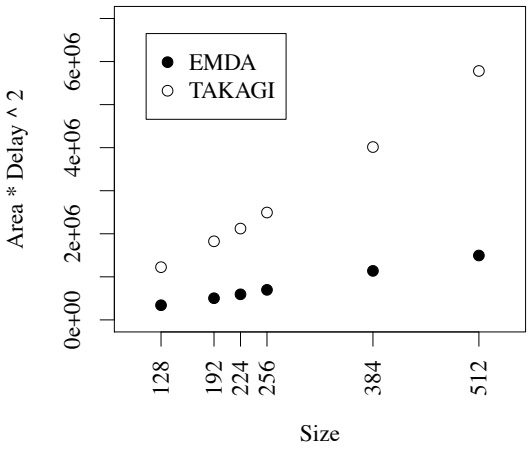

Figure 3.19: Area * Delayˆ2

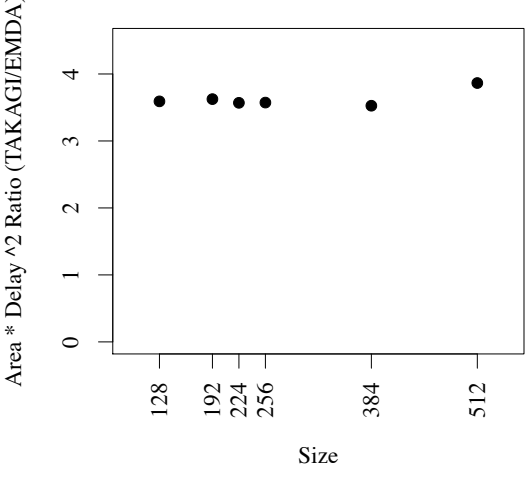

Figure 3.20: Area * Delayˆ2 Ratio

CHAPTER 4

MODIFIED MONTGOMERY MODULAR INVERSE ALGORITHM

This chapter proposes a modified version of Kaliski's Montgomery modular inverse algorithm. The proof of correctness and hardware model are presented. The chapter concludes with simulation results.

4.1 The Algorithm

The algorithm is based on Kaliski's algorithm (algorithm 5). It introduces a modification in the last iteration in the first phase. The modification eliminates the subtraction step (step 24) at the end of algorithm 5. The hardware implementation uses carry-save format for its constant addition time and smaller area

requirement.

The algorithm uses four operands (U, V, S and R) in computation. It keeps track of the bitsize of U and V in u_size and v_size respectively. The difference ($u\_size - v\_size$) is saved in the δ parameter which is represented by a shift register D and a sign flag $s \in \{-1, 1\}$ to reduce the cycle time ($\delta = s \times \log_2(D)$). The algorithm is shown in algorithm 7. Table 4.1 shows a comparison between the Algorithm 7 and the one given in [19].

Input: A, M, n; Where $0 < X < M$, M is odd prime, $n = \|M\|$
Output: $Z = A^{-1}2^n \ mod \ M$

1 Algorithm First Phase:
2 $U = M$; $V = A$; $R = 0$; $S = 1$; $K = 0$;
3 **while** $U > 0$ **do**
4 **if** $u_0 = 0$ **then**
5 $U = U/2$;
6 $S = 2S$;
7 **else if** $v_0 = 0$ **then**
8 $V = V/2$;
9 $R = 2R$;
10 **else if** $V > U$ **then**
11 $V = (V - U)/2$; $S = R + S$; $R = 2R$;
12 **else** /* $U \geq V$ */
13 $U = (U - V)/2$; $R = R + S$; $S = 2S$;
14 **end if**
15 $K = K + 1$;
16 **end while**
17 Algorithm Second Phase:
18 **for** $i = 0$ **to** $K - n$ **do**
19 **if** $s_0 = 0$ **then** $S = S/2$;
20 **else** $S = (S + M)/2$;
21 **end for**
22 $Z = S$;
23 **return** Z;

Algorithm 7: Modified Kaliski Modular Inverse Algorithm

Table 4.1: Comparing Algorithm 7 with [19]

Algorithm	Algorithm 7	[19]
Final Result	$A^{-1}2^n \ mod \ M$	$A^{-1}2^{2n} \ mod \ M$
Number Representation	Carry-Save	Signed-Digit
Sign Detection	Multiple bits per iteration	Single bit per iteration

4.2 Mathematical Analysis

4.2.1 Proof of Correctness

The algorithm maintains the following congruence relations:

$$A \times R \equiv -U \times 2^K \ mod \ M \tag{4.1}$$

$$A \times S \equiv V \times 2^K \ mod \ M \tag{4.2}$$

This can be verified by induction:

- At initialization $U = M$, $K = 0$ and $R = 0$ therefore equation 4.1 holds. $V = A$, $K = 0$ and $S = 1$ therefore equation 4.2 holds. Both equations hold after initialization regardless of the values of A and M.

- When U is even, the algorithm performs $U = U/2$ (step 5), $S = 2S$ (step 6) and $K = K + 1$ (step 15). For equation 4.1, U is divided by 2. This is compensated by incrementing the value of K, keeping the equation holding. At the same time S is multiplied by 2 to keep equation 4.2 holding.

- Similarly both equations hold when V is even.

- When $V > U$, the algorithm computes $V = (V - U)/2$, $S = R + S$, $R = 2R$

(step 11) and $K = K + 1$ (step 15). Theses steps multiply both sides of equation 4.1 by 2 and change the left hand side of equation 4.2 to $A \times (R + S) = A \times R + A \times S$ and the right hand side to $(V - U)/2 \times 2^{K+1} = V \times 2^K - U \times 2^K$. By using equation 4.1 and equation 4.2 as the induction hypothesis, the left hand side changes to $-U \times 2^K + V \times 2^K$ which is the same as the right hand side. Therefore, equation 4.2 also holds.

- When $U \geq V$, the algorithm computes $U = (U - V)/2$, $R = R + S$, $S = 2S$ (step 13) and $K = K + 1$ (step 15). Theses steps multiply both sides of equation 4.2 by 2 and change the left hand side of equation 4.1 to $A \times (R+S) = A \times R + A \times S$ and the right hand side to $-(U-V)/2 \times 2^{K+1} = V \times 2^K - U \times 2^K$. By using equation 4.1 and equation 4.2 as the induction hypothesis, the left hand side changes to $-U \times 2^K + V \times 2^K$ which is the same as the right hand side. Therefore, equation 4.1 also holds.

After every iteration, the algorithm reduces the magnitude of U or V until $U = V = GCD(A, M) = 1$. At this point $R \equiv -A^{-1}2^K \ mod \ M$ and $S \equiv A^{-1}2^K \ mod \ M$ according to equations 4.1 and 4.2. The algorithm then computes $U = (U - V)/2$, $R = R + S$, $S = 2S$ (step 13) and $K = K + 1$ (step 15) making $U = 0, V = 1, R \equiv 0 \ mod \ M$ and $S \equiv A^{-1}2^K \ mod \ M$.

According to the Theorem 2 in [13], $n < K < 2n$. The second phase reduces the value of S to the equivalent of $A^{-1}2^n \ mod \ M$.

4.3 Hardware Description

Registers

The implementation uses carry save format for the registers in order to have a constant cycle time independent of the operand sizes. All the operands (U, V, S and R) are represented by two registers (sum and carry).

According to Theorem 1 in [13], the intermediate values of U, V, S and R are between 0 and $2M - 1$. Thus, the lower bound of register sizes is $n + 1$. Being signed, an extra sign bit is added. In addition, the size is expanded to $n + 3$ to overcome the possibility that the sum and carry components can be greater than the theoretical value $2M - 1$.

The K operand is implemented as a shift register of size $2n + 1$ with a single bit set to 1 and all other bits are 0. The value of K is the position of the 1 bit in the shift register. The register is shifted one bit position to the left after every iteration in the first phase of the algorithm. In the second phase, it is shifted to the right until the bit at position n is 1 signaling algorithm termination.

Compressors

The operations in the algorithm are addition and subtraction. They are performed using three 4-2 compressors (Comp_U, Comp_V and Comp_RS). Comp_U and Comp_V compute the values of U and V respectively. Comp_RS is dedicated to perform operations on R and S in the first phase of the algorithm. Then it is used in the second phase to perform $(S + M)/2$ (step 20).

Sign Detection

In order to detect the sign of a number represented in carry-save, its sum and carry components have to be added with full carry propagation. To overcome this, the implementation performs the addition in multiple cycles. In each cycle, a window of size w is added from the sum and carry components starting at the most significant w bits. If the results is conclusive, the sign is returned. Otherwise, the window is shifted to the right. The process is repeated until a conclusive result is found or the window reaches the least significant w bits of the components.

When estimating the sign by using a window of few most significant bits, there is a possibility of a carry bit from the adding the least significant bits. Sign detection is conclusive if this carry bit does not flip the sign bit. The carry bit can flip the sign bit in two cases:

- When most significant bit is 0 and all other bits are 1.
- When all bits are 1.

The first case is not possible because it incurs an overflow. Therefore, the second case is the only inconclusive case where the window must be shifted right.

The sign detection algorithm is shown in algorithm 8.

4.3.1 The Datapath

The datapath is a direct implementation of the algorithm. The U, V path is separate from the R, S path. When the algorithm does a division by 2 (steps

Input: $A, size, w$; Where A is in carry save ($A = A_c + A_s$); $\|A\| \leq size$; w is the number of bits in the window
Output: $sign, newSize$

```
1  done = 0; sign = −1; y = size − 1; x = size − w; newSize = size;
2  (c_out, sum) = A_c[y : x] + A_s[y : x];
3  if sum[y] = 0 then
4      sign = 1;
5      done = 1;
6  else
7      for i = y − 1 to x do
8          if (sum[i] = 0) then
9              sign = −1;
10             done = 1;
11         end if
12     end for
13 end if
14 while done = 0  and  x ≥ w do
15     y = y − w;
16     x = x − w;
17     newSize = newSize − w;
18     (c_out, sum) = A_c[y : x] + A_s[y : x];
19     if c_out = 1 then
20         sign = 1;
21         done = 1;
22     else
23         for i = y to x do
24             if (sum[i] = 0) then
25                 sign = −1;
26                 done = 1;
27             end if
28         end for
29     end if
30 end while
31 return sign, newSize;
```

Algorithm 8: Find Sign Algorithm

5, 8, 19 and 20), the operands are shifted then applied to the compressor. This removes the need to correct the output of the compressor. Any bits lost due to shifting is supplied back through the compressor carry inputs.

When both U and V are odd, the datapath initiates find sign operation to find the sign of $U - V$. The find sign component selects the maximum of U_size and V_size using the values of s and D. It then creates two copies of s and D. Every iteration of find sign changes one copy of s and D to represent an increment in δ and changes the other copy to represent a decrement. After every iteration the size is decremented as well. At the end, if the result is positive or zero, U_size is updated and the incremented version of s and D is saved. Otherwise, V_size is updated and the decremented version of s and D is saved.

The datapath diagram is shown in Figure 4.1 and Figure 4.2.

4.4 Implementation Results and Comparison

4.4.1 Number of Cycles

Our modified Kaliski algorthim (algorthim 7) was simulated in Java and compared to the original Kaliski algorithm (algorithm 5) and the one proposed by Savas [19]. The simulation was done for 10000 random inverse operations with moduli sizes in the range 160 - 512 bits. The same test cases were applied for both algorithm 5 and algorithm 7. For the same test conditions, Savas has reported an average number of cycles of $12.34n$ [19]. The results are shown in Figure 4.3. The figure

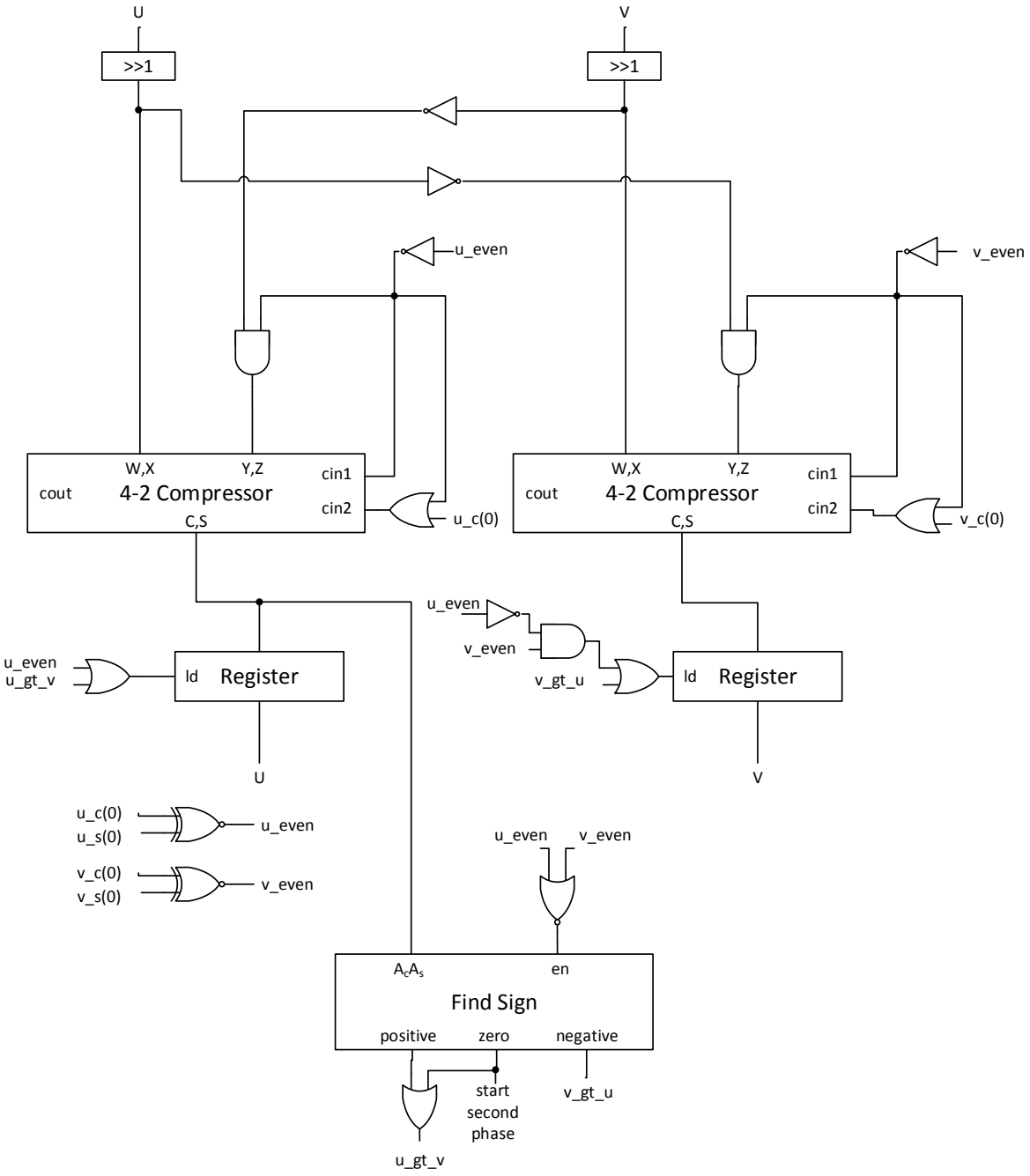

Figure 4.1: U, V Path

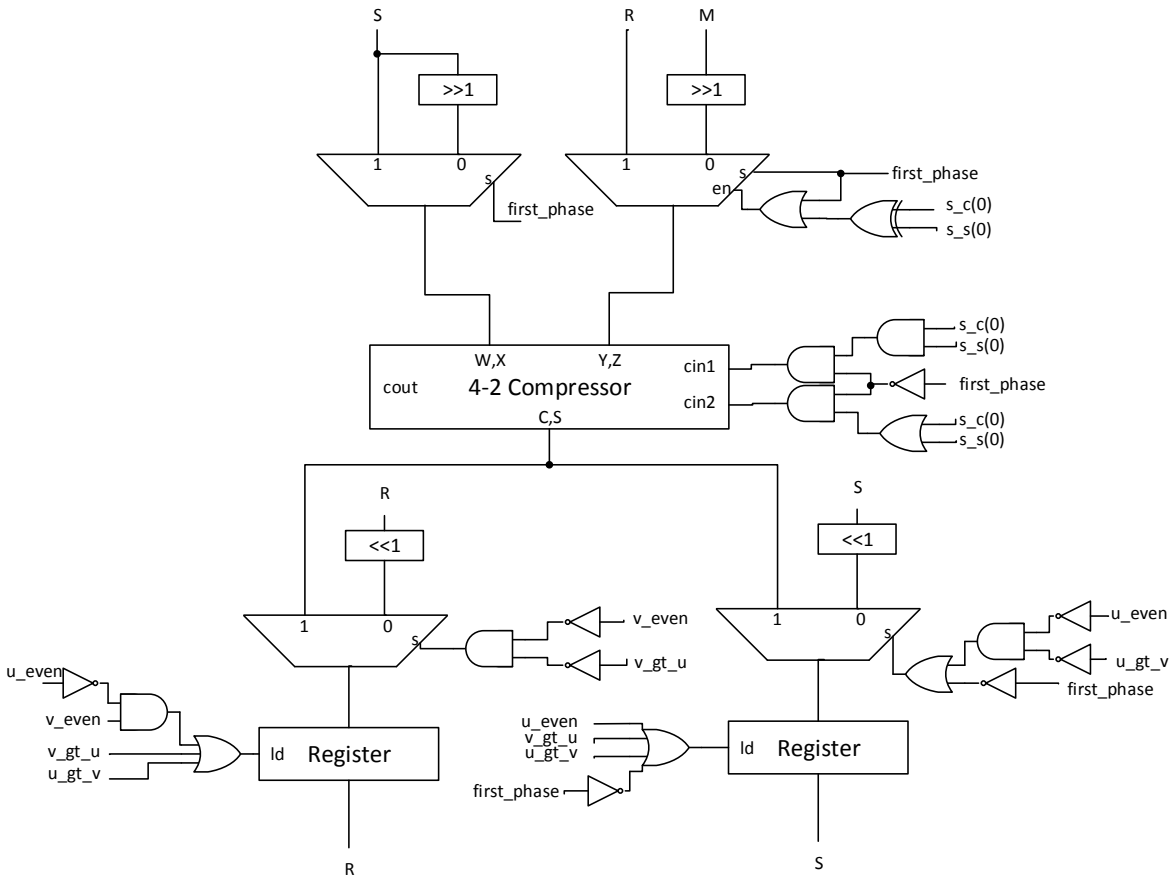

Figure 4.2: R, S Path

shows that algorithm 7 has a number of cycles close to algorithm 5 which does not consume any extra clock cycles since it uses carry propagate addition. As the window size increases, the sign estimation accuracy improves and the average number of extra cycles reduces. Figure 4.4 shows the average number of cycles of algorithm 7 and Savas's algorithm [19] normalized to that of Kaliski (algorithm 5). We recommend a window size of 2 or 4 since they give an average number of cycles of $2.89n$ and $2.39n$ respectively, i.e a saving of 76% and 80% compared to [19], while at the same time do not have a significant effect on area and cycle time.

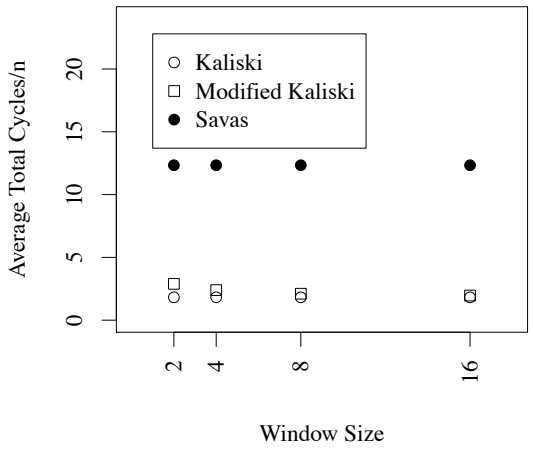

Figure 4.3: Average Number of Cycles/n

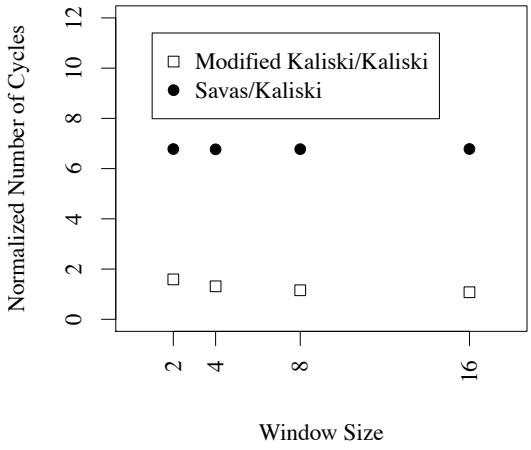

Figure 4.4: Number of Cycles Normalized to Kaliski

CHAPTER 5

CONCLUSION

In this thesis, two algorithms were presented. The first computes modular division while the second computes the Montgomery modular inverse. Proof of correctness was provided for both algorithms.

The division algorithm is based on the extended binary GCD algorithm. It replaces the loop ending condition (comparison against 0) with bit size comparison. This reduced the critical path delay and introduced extra clock cycles. The implementation consumes less area and has lower critical path delay than previous implementations.

The inverse algorithm is an implementation of a modified version of Kaliski's modular inverse algorithm. The modification eliminates one step in the original algorithm. The implementation does value comparison by doing constant delay subtraction and detecting the sign of the result in multiple clock cycles. This implementation improves over previous work by using a window of bits to find the sign, reducing the overall number of iterations.

Carry-save format was used in both implementations in order to make the critical path delay independent of the operand sizes. Carry save is chosen over binary signed digit for its lower area per bit. The issue related to right shifting the components of carry save numbers was resolved.

Future Work

- The division algorithm can be improved by trying to reduce the number of extra clock cycles. This can be achieved by finding more situations to safely reduce the a_size.

- The comparison step in the extended binary GCD algorithm is the most time consuming step. Doing the comparison in multiple cycles can be compared to current division implementation in terms of number of iterations and critical path delay.

- The inverse algorithm uses normal carry propagate adder when detecting the sign. Other types of adders (e.g. carry look ahead and carry select) can be used to improve the accuracy when finding the sign while maintaining the critical path delay.

- The two algorithms have similar properties. Combining them in one implementation could be a research direction, especially if the division was performed using a direct implementation of the extended binary GCD algorithm.

REFERENCES

[1] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, Nov 1976.

[2] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *Information Theory, IEEE Transactions on*, vol. 31, no. 4, pp. 469–472, Jul 1985.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[4] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comp.*, vol. 48, no. 177, pp. 203–209, 1987. [Online]. Available: http://dx.doi.org/10.2307/2007884

[5] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[6] M. Aydos, T. Yanik, and C. Koc, "High-speed implementation of an ecc-based wireless authentication protocol on an arm microprocessor," *Communications, IEE Proceedings-*, vol. 148, no. 5, pp. 273–279, Oct 2001.

[7] E. Berlekamp, *Algebraic coding theory.* New York: McGraw-Hill, 1968.

[8] T. Jebelean, "Comparing several gcd algorithms," in *Computer Arithmetic, 1993. Proceedings., 11th Symposium on*, Jun 1993, pp. 180–185.

[9] L. Tawalbeh, A. Tenca, S. Park, and C. Koc, "A dual-field modular division algorithm and architecture for application specific hardware," in *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, vol. 1, Nov 2004, pp. 483–487 Vol.1.

[10] M. Kaihara and N. Takagi, "A hardware algorithm for modular multiplication/division," *Computers, IEEE Transactions on*, vol. 54, no. 1, pp. 12–21, Jan 2005.

[11] G. Chen, G. Bai, and H. Chen, "A new systolic architecture for modular division," *Computers, IEEE Transactions on*, vol. 56, no. 2, pp. 282–286, Feb 2007.

[12] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comp.*, vol. 44, no. 170, pp. 519–521, 1985. [Online]. Available: http://dx.doi.org/10.2307/2007970

[13] J. Kaliski, B.S., "The montgomery inverse and its applications," *Computers, IEEE Transactions on*, vol. 44, no. 8, pp. 1064–1065, Aug 1995.

[14] E. Savas and C. Koc, "The montgomery modular inverse-revisited," *Computers, IEEE Transactions on*, vol. 49, no. 7, pp. 763–766, Jul 2000.

[15] G. de Dormale, P. Bulens, and J.-J. Quisquater, "An improved montgomery modular inversion targeted for efficient implementation on fpga," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, Dec 2004, pp. 441–444.

[16] M. Naseer and E. Savas, "Hardware implementation of a novel inversion algorithm," in *Circuits and Systems, 2003 IEEE 46th Midwest Symposium on*, vol. 2, Dec 2003, pp. 798–801 Vol. 2.

[17] R. Lórencz and J. Hlaváč, "Subtraction-free almost montgomery inverse algorithm," *Information Processing Letters*, vol. 94, no. 1, pp. 11 – 14, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019004003692

[18] R. Deng and Y. Zhou, "Improvement to montgomery modular inverse algorithm," *Computers, IEEE Transactions on*, vol. 55, no. 9, pp. 1207–1210, Sept 2006.

[19] E. Savas, "A carry-free architecture for montgomery inversion," *Computers, IEEE Transactions on*, vol. 54, no. 12, pp. 1508–1519, Dec 2005.

Vitae

- Name: Mohamed Abuobaida Mohamed
- Nationality: Sudanese
- Date of Birth: 23/8/1989
- Email: *mho000@hotmail.com*
- Permenant Address: Khartoum, Sudan

- BSc. King Fahd University of Petroleum and Minerals (KFUPM), Dhahran. Computer Engineering Department (COE). Graduated June 2010. GPA 3.78 out of 4.
- Electives: Design and Modelling of Digital Systems, Database Systems, Design and Analysis of Algorithms and Computer System Performance Evaluation.
- Achieved CCNA Discovery: Networking for Home and Small Business and CCNA Discovery: Working at a Small-to-Medium Business or ISP.
- Placed 2^{nd} in the 6^{th} Annual International Microelectronics Olympiad held by Synopsys in Armenia in October 2011.

- Participated in the 3^{rd} Scientific Conference for Students of Higher Education in Saudi Arabia in May 2012.